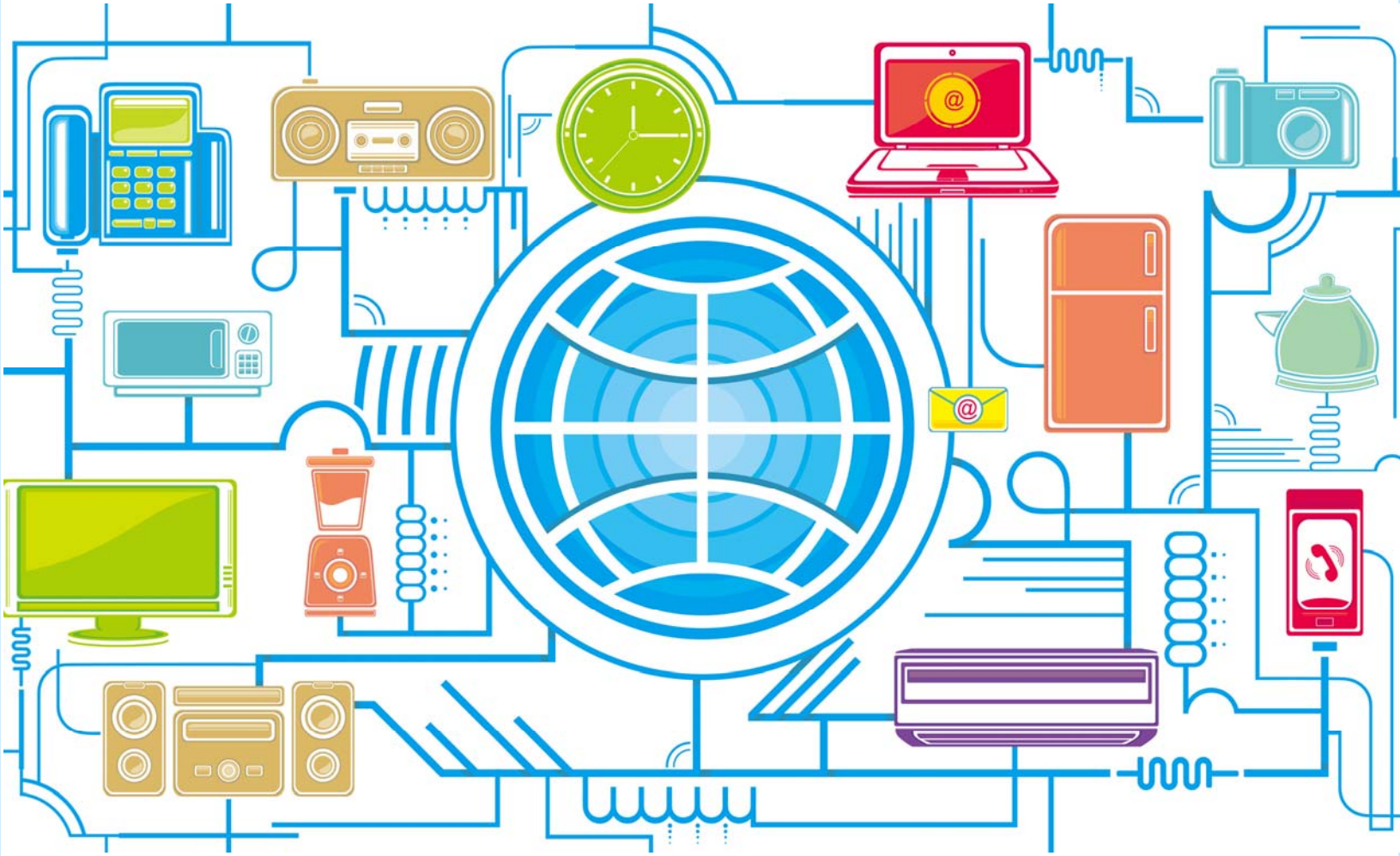
SHOWCASING THOUGHT LEADERSHIP AND ADVANCES IN SOFTWARE TESTING

Testing Embedded Systems and the Internet of Things



Software Testing's New Frontier

Michael Hackett, LogiGear Corporation

Testing Strategy for the IOT
Jon Hagar

Why you Need a Software Specific Test Plan
Philip Koopman

Proven Practices for Testing Embedded Software
Andrey Pronin

August 2014 | Volume VIII | Issue 3

Letter from the Editor

Editor in Chief
Michael Hackett

Managing Editor
Brian Letwin

Deputy Editor
Joe Luthy

Worldwide Offices

United States Headquarters
4100 E 3rd Ave, Ste 150
Foster City, CA 94404
Tel +01 650 572 1400
Fax +01 650 572 2822

Viet Nam Headquarters
1A Phan Xich Long, Ward 2
Phu Nhuan District
Ho Chi Minh City
Tel +84 8 3995 4072
Fax +84 8 3995 4076

Viet Nam, Da Nang
VNPT Tower, F/l 7 & 8
346 Street 2/9
Hai Chau District
Da Nang
Tel +84 511 3655 33

www.LogiGear.com
www.LogiGear.vn
www.LogiGearmagazine.com

Copyright 2014
LogiGear Corporation
All rights reserved.
Reproduction without permission is prohibited.

Submission guidelines are located at http://www.LogiGear.com/magazine/issue/news/editorial-calendar-and-submission-guidelines/

Testing Embedded systems and testing the Internet of Things could each have their own issue of LogiGear magazine. But these days they are referred to presupposing knowledge of the other, so we thought it a good idea to tackle the two together in this issue to give a broad understanding of the landscape as well as help you get started testing smarter and more informed.

The Internet of Things (IoT) is getting more and more press and attention every day. There is nothing new about the idea, everyone knew it was "the future." But that future is here. Efficient wireless protocols, sensors to sense everything, and cheaper processors are making the future now. The number of things with embedded systems is already staggering, and it's estimated there will be tens of billions of embedded system devices connected by 2020.

Although often used interchangeably, embedded systems are *not equal* to the IoT. The difference is that embedded systems do not have to be *connected* to anything while the *thing*s in the IoT are connected via the internet. Devices now do things like monitor your glucose level and send a constant stream of data to your doctor. Others count the steps you take, your refrigerator can order milk when the carton is near-empty. and self-driving cars are in prototype.

My early embedded systems testing experience focused on medical device testing and mobile communication/mobile phone testing. In each case, as is common with embedded systems, the functionality was important but very limited, the UI was very limited or non-existent. The testing started with validating requirements. We then made models of expected behaviors and tested as thoroughly as we had time to do.

The IoT is a big leap ahead from closed system embedded devices with limited functionality and no connectivity. Even for embedded system experienced testers connectivity is a big change. There are potentially large data being produced, connection to other devices and APIs to other services. This opens the door to such things as interoperability, security and performance issues not normally seen in embedded system testing. All with speed of delivery demands in an industry with few and often competing standards.

As is always the case with new or more pervasive technologies: how do you test it? What are the new testing concerns? What skills do you have to learn to respond to this staggering growth? That is what this issue begins to explore.

In this issue I discuss the landscape of the Internet of Things with Joe Luthy; Jon Hagar gives expert recommendations for testing the IoT; I explain new areas for traditional testers working in the IoT; Phillip Koopman suggests that those performing embedded testing have software specific test plan; Auriga's Andrey Pronin writes about the importance of a highly organized testing process and Essensium/Mind's Arnout Vandecappelle reminds us that team collaboration is essential for testing embedded systems. Welcome to the IoT.

In this Issue

4 IN THE NEWS

5 INTERVIEW WITH MICHAEL HACKET: THE IOT LANDSCAPE

Michael Hackett and Joe Luthy, LogiGear Corporation

Quick start understanding of the Internet of Things from a development team perspective

10 TESTING STRATEGY FOR THE IOT

Jon Hagar

Experience report: A guide through the challenges and risks of testing the IoT

15 SOFTWARE TESTING'S NEW FRONTIER

Michael Hackett, LogiGear

What you need to know for testing in the new paradigm

21 BLOGGER OF THE MONTH: WHY YOU NEED A SOFTWARE SPECIFIC TEST PLAN

Phillip Koopman

Experience-based recommendations to test the brains that drive the devices

23 PROVEN PRACTICES FOR TESTING EMBEDDED SOFTWARE

Andrey Pronin, Auriga

Build the right test platform including infrastructure, virtual lab and process

30 TEAM BASED TESTING

Arnout Vandecappelle, Essensium/Mind

Getting everyone involved lets you shorten the loop by testing early

34 GLOSSARY

Some of the terms used when discussing testing embedded systems

In the News

Agile Testers Must Develop New Mindset: Janet Gregory

Testers have a vital role to play in the agile world, but in order to do so, they need to develop a distinct mindset. Speaking at a workshop hosted by IndigoCube, Janet Gregory, a leading international expert in agile testing, outlined the characteristics of an agile tester.

"The agile tester no longer sees herself as someone whose main purpose is to identify bugs or ensure that specific requirements are met. Agile testers are thinkers and take a much broader view, asking: 'What can I do to help deliver the software successfully?' says Gregory.

"The ideal agile tester is analytical, curious, observant and critical – and combines a wide cross-functional knowledge with deep skills in the discipline of testing."

Maersk Begins Software Testing on Five Vessels

Global container shipping company Maersk Line has started pilot testing of a maritime software tool, ShipManager, on five of its vessels.

Norway-based shipping classification society DNV GL has developed the software, which is said to deliver increased process efficiency and improved access to and analysis of information.

Maersk Line has selected the application as the preferred solution over 26 other ship management platforms for installation onto its self-owned fleet of 250 vessels in 2015.

150 Jobs Coming to South Bronx at Software Testing Center

A piece of Silicon Alley is coming to The Bronx. An IT job training nonprofit plans to open a massive software testing center in Port Morris this fall that will bring 150 jobs to the area.

Per Scholas, which is based in The Bronx, and the IT consulting company Doran Jones plan to open the roughly $1 million, three-story, 90,000-square-foot software testing center at 804 E. 138th St., near Willow Avenue.

Starting wages for the jobs will be $35,000 with benefits, and 80 percent of hires will be Per Scholas graduates.

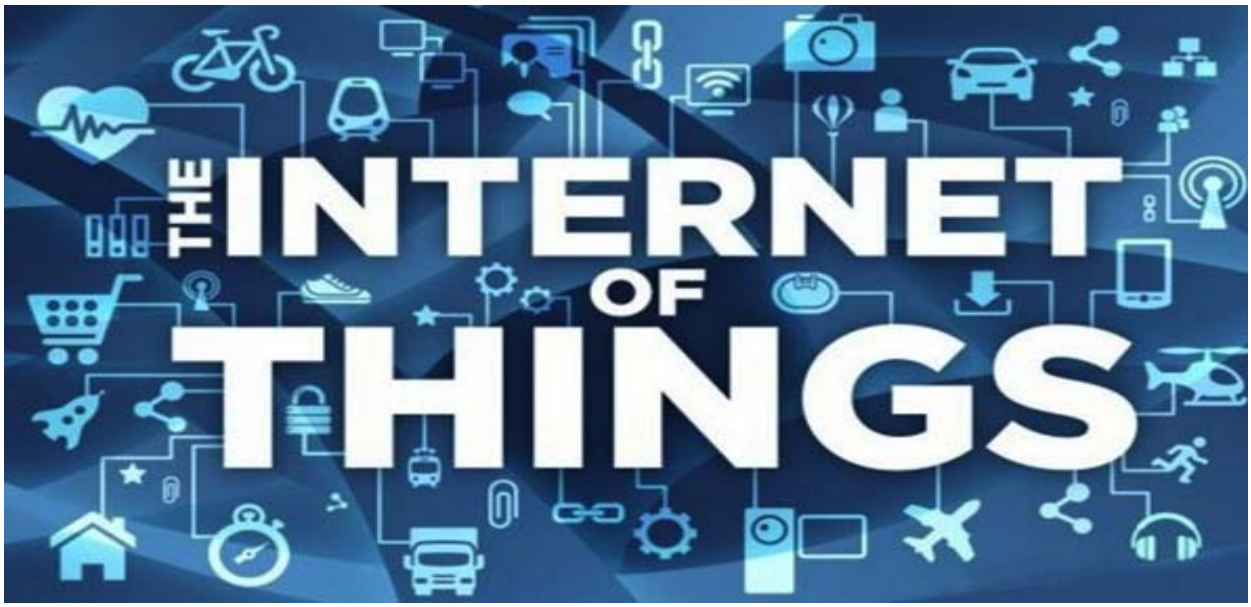
"A lot of folks who are based in the community will actually get access to the jobs, which they wouldn't ordinarily," said Co-CEO Keith Klain.

Feature

An Interview with Michael Hackett

Quick start understanding of the Internet of Things from a development team perspective

By Joe Luthy, LogiGear Corporation



When I hear the word internet I tend to think about information, interaction and communication as it relates to people using and controlling all sorts of devices, but it's much more dynamic than that. More and more, devices, computers and even interconnected computer systems are interacting and communicating with each other to assist us humans, and in a lot of cases, control our activity. The *Internet of Things* (IoT) is drastically changing the traditional view of the internet.

For this article I sat down with Michael Hackett to pick his brain about the IoT. Since his early experience testing embedded devices, Michael has been actively looking at the space and helping companies create and develop ways to test the systems that are becoming too critical to fail.

JL: Isn't "The Internet of Things" just a conglomeration of embedded systems?

MH: Embedded systems are the foundation of the IoT. Embedded devices and systems have been around for a very long time—since the beginning of computers. It is widely believed the first *internet appliance* or first *thing* in the IoT went online in 1982, the Networked Coke Machine. Four students at Carnegie Mellon University used a sensor to find out remotely if a coke machine had coke or was empty – sounds super simple by today's standards. It was 30 years ago.

An *embedded* device could simply be a chip in a small system or a very, very large system with a narrow scope of specific tasks. They range from controls in things like large factory automation systems or airplanes to narrower focused medical devices or mobile phones, to light switches and door locks. These systems can have sensors and electrical and/or mechanical parts. They may even connect to a server through the internet and up- and down-load data.

The IoT can grossly be described as interacting, embedded systems. More efficient wireless protocols, improved sensors, cheaper processors, and creativity have come together to produce a flood of new products where embedded devices are connected through the internet and can be remotely controlled— for example, from your mobile device—and even control each other. Using a program on your phone to turn your lights a few minutes before you arrive home is the IoT at work.

JL: What's the direction of the IoT?

MH: The space is growing rapidly. Vast webs of interconnected devices are already communicating machine to machine (M2M) for specific purposes while at the same time creating and then using big sets of data and cloud services that provide products and services to all kinds of people and organizations from individuals to power companies.

What makes this time special is the newness of so many products and the speed at which they are being delivered to consumers.

It was not too long ago that someone on a heart monitor had to stay in the hospital. Now, people go about their daily life with implanted heart monitors. Data is uploaded wirelessly, and in if an emergency happens, all kinds of notifications and emergency calls will be triggered automatically through Wi-Fi or cellular service. This isn't new. What is new is the speed at which these things are entering the market.

The IoT is also shrinking ideas of what we think of as embedded systems. Early devices typically had little processing power and little memory, and a rudimentary UI. The devices being developed today are smaller with even tinier batteries, non-existent UIs, and more lines of code. These things make programming and testing interesting, challenging and difficult all at the same time. Even the few functions they perform probably don't run on MS Windows Embedded. It probably has a tiny proprietary operating system that will have a steep learning curve for programmers and testers to climb.

JL: Speaking of Windows, is Microsoft in the position to own the OS in the space?

MH: This is the hot question today—Who will own the IoT? Cisco? Intel/WindRiver? Apple/iOS? Google/Android? Or will it be a company like GE, that is seemingly a step ahead of everyone on sensors, devices and consumer products?

On one hand you could say it doesn't matter. But really, it does. When it comes to developing products, programmers and testers work and build in separate environments. Specifically with the IoT, these environments or platforms—clearly— have to talk to each other. Some companies don't make it easy for products to communicate with each other. If I need a heating unit that can quickly communicate with mobile devices, web browsers, the local power company and the national weather service API in the cloud, there can be problems. It would be far easier if everyone used the same internet protocols and messaging.

Also, programmers use platform tools and have favorite programming languages. Testers have knowledge of common operating systems and tools that work on some platforms and not others. Proprietary or uncommon operating systems will slow teams down at the start. Which platform will teams be more knowledgeable of and have access to more tools: Windows Embedded 8, iOS, WindRiver VxWorks or embedded Linux?

Currently, the rush into health and smart home devices seems to favor Android and iOS due to their dominance in the mobile device market.

There are many, many big players— all in fierce competition. I only care about programmers and testers that need to have quick ramp-up on platforms that will cooperate with each other and provide interoperability. These programmers and testers will have to have access to tools, particularly test automation tools, that will not slow them down under pressure to get products to consumers.

I mentioned Microsoft and Windows Embedded 8 not because Microsoft is a sexy company again, but because their Embedded OS has quick and easy communication to a wide variety of enterprise services, all kinds of backend services, desktop, tablet and smart phone integration. Plus, Microsoft has spent decades making it possible for their platforms communicate with other systems, network protocols and, a ton of tools. These are important interoperability considerations for companies, programmers and testers on the cusp of the world full of connected embedded systems.

JL: You mentioned "standards"; are there any standards?

MH: No. Not yet and I doubt there will be soon. Right now there are competing attempts at standards. There is the Internet Engineering Task Force (IETF www.ietf.org) for embedded devices. At their event, members of the alliance focused on device and protocol interoperability for building the IoT.

There's the IPSO Alliance that is seeking to establish the Internet Protocol as the network for the connection of Smart Objects. The alliance includes chip, sensor and network makers plus Oracle, SAP, National Instruments, Fujitsu, and a few more. Competing with that group is the Open Interconnect Consortium with Intel, Dell and Samsung, Broadcom (BRCM) and Wind River.

Of course Apple and Google are each obviously interested in creating a dominant standard. The most likely scenario is there will be a lot of competition early and more fragmentation, but eventually there will be have to be some standards.

JL: Does it matter?

MH: Well, for testing it certainly does! The whole purpose of the IoT is interconnectivity. Not having standards presents a myriad of testing issues.

JL: What will be the primary wireless connection?

MH: Good question. Different technologies will be used in different situations. Some machines will use TCP/IP and Wi-Fi. Bluetooth and cellular (3G and 4G) will also be common. For example, automobiles and their various devices and systems use cellular to communicate and that looks pretty stable. TCP/IP and Wi-Fi will probably wind up being most common for fixed location devices.

Bluetooth is being used in all kinds of IoT applications. It is especially useful with *low* and *very low power* products for close proximity communication. RFID (radio frequency Identification) tags and NFC (near field communication) will also continue to have a presence in the IoT.

Each of these has limitations, problem areas and unique error conditions. These are technologies that will be helpful to learn about for programmers and testers to do their jobs more effectively.

JL: Interesting. You have mentioned M2M a few times. What is special about that?

MH: M2M is actually the foundation for the IoT. Machines communicating with other machines on a much larger scale is IoT. M2M communication is enabling connected devices to exchange and use information without human involvement.

A common example of the use of M2M, without calling it M2M is in Smart Homes. Motion sensors communicate with light switches and cameras. Door locks and window sensors communicate with alarm sensors. All this happens without human intervention. M2M is common in industrial applications, removing humans from hazardous work environments.

JL: Some people discuss the IoT together with other current tech trends like Cloud APIs and Big

Data. How are they related?

MH: For many companies, that is the goal. This gets to the core of the IoT—to connect devices, not only to control them, but to integrate and do things with them. Having sensors constantly report conditions, build massive data, use some cloud service to correlate the data and then predict behavior that will change the response of any number of devices. This is already common in public and industrial application. From traffic controls to power grids, the IoT builds big data, uses cloud APIs, recognizes issues or predicts behavior and adjusts, adapts devices as a result.

There are a wide range of test issues that come up for these situations: very different technologies to test, different areas of expertise to build, integration and interoperability issues, latency, performance, load, and clearly, security.

JL: Security must be a big problem.

MH: There have been some very public, high profile security problems like baby monitors and insulin pumps getting hacked. The baby monitor breach highlights the fear that anyone can hack into your home security system. Think about this: If you can access a camera in your house to see if anyone is there, a thief can hack into the camera and get the same information. That pretty much negates the purpose of having a home security system.

The possibility that someone could hack into your insulin pump and turn it off or over-dose you does not need explanation; it is dangerous. Reports say security problems are abundant on devices. When you open up these devices to the internet, you are opening it up to all kinds of danger.

Some of the organizations trying to set standards are focusing on the need for more secure communication as the whole reason for standards. Although security testing has typically been separate from functional testing, a big part of IoT product development will be security testing.

JL: With embedded systems, we often hear the phrase "real time system." How are these different?

MH: Real time is a category of devices of systems that have time critical functionality. Anti-locking brakes are a typical example. The brake embedded system can't have a delay. I do not know how fast the anti-locking brake response must be, but it must be in mille-seconds.

It is typically safety critical, mission critical services that have to happen without buffering or queueing delay. These systems have advanced algorithms for scheduling and tend to have very narrow functional focus and strict latency requirements to meet. Adding more functionality to these devices: connectivity, data collection, security will often have a performance hit.

JL: So what's the take away from this?

MH: More learning, more testing, new discoveries and then even more testing will have to be done.

EMBEDDED SYSTEMS RUN THE WORLD

10 BILLION Processors shipped annually

98% Used in Embedded Systems

- **40** at Work — PC, printers, scanners, phones
- **40** at Play — Mobile devices, ATMs, GYM equipment
- **70** in car — Ignition, ABS, door locks, engine control
- **80** at Home — TV, DVR, games, appliances

250+ Daily Interactions

Computer on Wheels

- up to 100 microprocessors
- 1GB of Software
- 100 Million Lines of Code

Feature

Testing Strategy for the IOT

Experience report: A guide through the challenges and risks of testing the IoT

By Jon Hagar

Embedded software has been around for years, going back to the dawn of computers. Traditionally we tested these devices in isolation and did not worry about user interfaces (if there was one) or things such as internet connectivity. The connectivity of devices started not long after the internet's arrival. However, in recent years the so called "Internet of Things" (IoT) has become of more importance and certainly more newsworthy as its use is growing rapidly.

The acronym, IoT, identifies the advanced connectivity of devices, systems and services beyond the classic web and network connections of information technology systems (IT and PCs). IoT includes a number of protocols, many devices environments, and even more applications. There are millions of IoT devices currently connected and predictions are that there will be nearly 26 billion devices or more [http://www.gartner.com/newsroom/id/2636073] by 2020. IoT connections include wired and wireless devices with approaches such as low power radio, Wi-Fi, Bluetooth and others. Many of these devices will use an IP address or a group connection through secondary IP addressable devices such as hubs, bridges and/or routers. We are putting IoT in our homes [Time Magazine, Vol 184, no 1, 2014], in health care, businesses, and everywhere else.

IoT devices will share the development and test issues found in embedded software systems as well as more traditional IT/Web systems. With increasing numbers of IoT devices and software projects, the need for testers and new testing approaches will also increase. Testers coming from these historic environments will face different testing challenges and bugs. This article outlines some starting points for those going into IoT testing and offers considera-

tions for those already testing IoTs. Testing is a large subject with many books and thousands of articles, so readers should follow the links and resources to continue their learning. Remember, no one can know it all, but there are great reference materials available in many forms.

Examples of Product Test Challenges and Risks that IoT Testers Face

Testers face both new and old potential problems (errors) in IoT devices. These include:

- Embedded functionality,
- Web provided functionality,
- Performance both of the network communication and internal computation,
- Security including privacy, autonomy and control,
- Smartness of the device and the user interface or of the software in some devices (may hide bugs),
- Architecture of the hardware and of software, means more configurations must be tested, e.g., Android fragmentation [http://opensignal.com/reports/fragmentation-2013/],
- Complexity of the software and system (means more bugs may be in the code hiding in the complexity),
- The devices may have large amounts of code e.g., smart phones now have 10-to-20 million lines of code (where errors can hide),
- Development time considerations, such as time to market pressure, which exists in IT and Mobile, will continue with IoT,
- Resource considerations such as limitations in: memory, processing power, bandwidth, battery life, etc.
- Unique environments the devices will be used in: hot, cold, wet, noise, at altitude, etc.

Many testers will be familiar with two or three of these issues but not the others. For example, many historic embedded software testers verified functionality and CPU timing issues yet did not worry about connectivity, performance, security, usability, or large amounts of code. Historic Web/IT testers worked these secondary items and did not worry about issues common in embedded systems such as: limited resources, unique hardware functionality, and high-risk, critical device control problems.

Additionally, I have heard project stories where historic embedded devices were "updated" with a network card or mobile connection. The embedded device was working so all the new testing focused only on the "new" connection. Could there be a problem with this line of thinking and how much would that cost the company? Consider the possible limitations of this simplistic initial testing and usage:

1. Security holes from the historic code may be missed.
2. Performance testing was CPU usage based and did not consider the impact of the connections, e.g., long waits (seconds versus mille- or micro seconds), loads, slow network, dropped connections, etc.
3. Viability and completeness of recorded data.
4. Usability of the system with the new connection.
5. Coupling impact from the new logic to existing functionality.

Certainly these challenges and risk are not the only ones IoT testers will face, but these are a start. And, once costs are examined with finding issues after a product is released, companies could lose a lot of profit.

A Start: IoT Strategy and Planning Impacts

A team should consider the implication of test strategies for both the new and re-hosted IoT device. I would start by obtaining and using IEEE1012 Verification and Validation standard [http://standards.ieee.org/findstds/standard/1012-2012.html]. Using this standard, I would assess device V&V test activities against my estimations of risk and determine an integrity level (defined in IEEE1012 and determine amounts and types of test activities). When dealing with a historic device, try analyzing white and black-box coverage levels (e.g., statement coverage, requirements coverage, performance analysis, etc.). When dealing with new devices, consider the product's risks, quality characteristics and

functionality. Finally, consider the strategy in light of allocated cost and schedule. The complete strategic information is reviewed with the stakeholders so that everyone agrees on the strategy before beginning test planning and design efforts.

Next Step: IoT Implication to test plans

Once there is agreement on test strategy, use it to guide the IoT software test plan. Here again, if you are new to IoT testing, continue with a refinement of the concepts from IEEE1012 to the next level of detail. Follow this planning with the test concepts, processes, and techniques from ISO29119. When using standards, tailor them to your local context, since a standard is only a basic beginning and not an end or best practice. A test organization that already has strong test practices and skilled testers might not need this standard since a skilled group can leverage their history and knowledge to start an IoT test plan. However, for test groups without much IoT history, I would analyze in more detail the testing that has been completed, look for error taxonomy information, determine what test concepts to include, and have a sound method for regression testing [http://www.logigear.com/magazine/issue/3350/].

Both new and historic IoT organizations should consider what test concepts and environment should be added to the test plan including:

- Risk-based testing [29119];
- Test attacks [Whittaker, *How to Break Software,* Hagar, *Software Test Attacks to Break Mobile and Embedded Devices*] to find risky bugs;
- Exploratory testing times and efforts;
- Required (regulatory) scripted testing and documentation [ISO29119-3];
- Test tools and automation needed;
- Test lab(s) set up;
- Test tours [Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, Whittaker] to use; and
- Test techniques to apply [ISO/IEC 29119-4].

An example IoT test lifecycle pattern for a test plan might look like:

- Strategy
- Plan
- Design with regression considerations
- Act (test)
- Report and document [ISO/IEC 29119-3]
- Repeat (within resource boundaries such as test team skill, cost, and schedule).

These activities might take day or hours, depending on the project context.

Finally, I find it is easy to forget things in test planning, so I like to use checklists to help me complete my planning, remembering that as soon as I start to execute my plan, the plan will change and I'll have to refer back to my strategy, plans and checklist frequently. A sample beginning checklist is given in table 1.

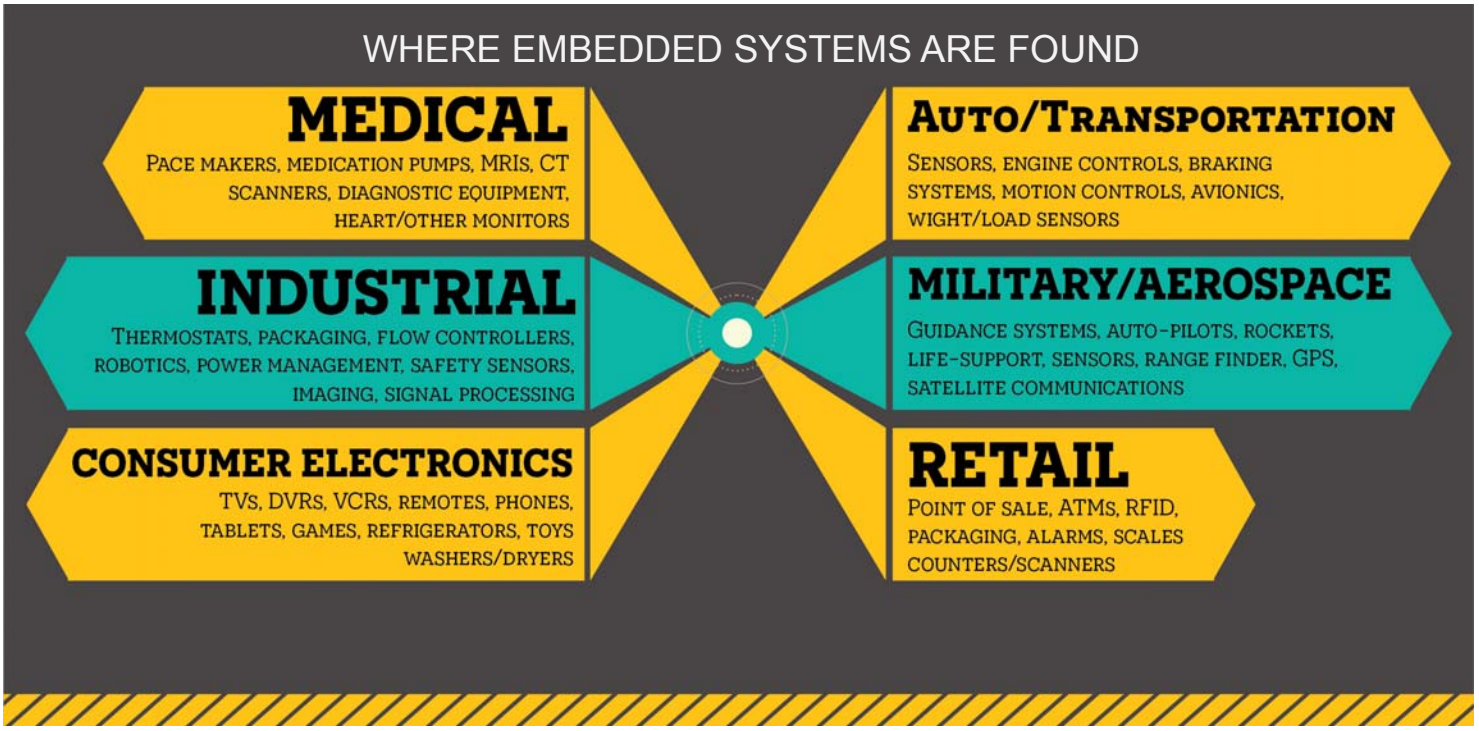
Test Tools Needed to Support IoT

When a tester says "test tools" everyone typically thinks automated test execution tool, and while this is part of the story when I say the word "tool", I mean anything that helps me to do better testing. Tools can be pieces of software, such as capture-playback tools, but a tool can also be a checklist, which supports manual testing.

I recommend white-box and black-box testing including analysis concepts such as static code analysis tools [Hagar, *Software Test Attacks to Break Mobile and Embedded Devices*] . These levels and approaches to testing allow testing, verification, and validation to be done throughout the lifecycle. Also, these combinations are complementary increasing the likelihood that errors will be found. Finally, many IoT embedded projects may benefit from the use of model and mathematical analysis which, in my experience, more progressive organizations will have the ability to use.

Table 1: Test Planning Checklist

1.	Is this a "new" device or a re-hosting of an older device to IoT
	New devices need full testing Re-hosted devices may just need testing of the IoT connection logic and regression testing including assessing performance changes
2.	Risk-based testing
	What are the functional risks? What are the non-functional risks? What are the risks of going to IoT connections? Are there any security concerns? Is the device small, big, critical, and does it have any safety factors a tester must consider, etc.?
3.	What data is going to be shared to the outside world (privacy?)
	Who receives the data (and who definitely should *not* receive the data)?
4.	For incoming data, what are the security threats and/or in useage safeguards?
5.	Is the device mobile, fixed, or mixed (testing will be different for each)?
6.	Where in the world is the device going to be used (localization issues)?
7.	Who are the possible users (human, other computers, hardware, sensors, etc.)?
8.	What is the complexity of the problem being solved?
9.	What does the company/groups error taxonomy history indicate for test plans?
10.	Are there regulatory, organizational and/or project test processes which must be adhered to?
11.	What are the verification and validation needs?
12.	What documentation is needed?
13.	What is the skill level of the team and testers?
14.	What test techniques are going to be used?


WHERE EMBEDDED SYSTEMS ARE FOUND
MEDICAL
PACE MAKERS, MEDICATION PUMPS, MRIS, CT SCANNERS, DIAGNOSTIC EQUIPMENT, HEART/OTHER MONITORS
AUTO/TRANSPORTATION
SENSORS, ENGINE CONTROLS, BRAKING SYSTEMS, MOTION CONTROLS, AVIONICS, WIGHT/LOAD SENSORS
INDUSTRIAL
THERMOSTATS, PACKAGING, FLOW CONTROLLERS, ROBOTICS, POWER MANAGEMENT, SAFETY SENSORS, IMAGING, SIGNAL PROCESSING
MILITARY/AEROSPACE
GUIDANCE SYSTEMS, AUTO-PILOTS, ROCKETS, LIFE-SUPPORT, SENSORS, RANGE FINDER, GPS, SATELLITE COMMUNICATIONS
CONSUMER ELECTRONICS
TVS, DVRS, VCRS, REMOTES, PHONES, TABLETS, GAMES, REFRIGERATORS, TOYS WASHERS/DRYERS
RETAIL
POINT OF SALE, ATMS, RFID, PACKAGING, ALARMS, SCALES COUNTERS/SCANNERS

Classic test execution automation will support many of the issues in IoT such as testing device configurations and capture/playback. Support by vendors for embedded, mobile, IoT testing has been increasing in recent years. Teams working on IoT are advised to conduct tool trade studies and searches to find the best candidate tools for their project's context.

The use of tools and automation does not mean that all testing should be automated. I find the advanced groups mix automated test execution and manual testing, particularly guided exploratory testing, with tours and attack patterns [Whittaker and me]. This follows the concept of having complementary approaches and activities to guide the testing. Complementary ideas would be reflected in test plans and designs.

Recommend Tester Knowledge and Skills

More than test processes or tools, skilled testers are needed for IoT. *Software testing is practiced well when the knowledge and skill of the person doing the work determines the effectiveness of the effort.* A project can have good practices and the right tools, but unless there are skilled people to drive these efforts, good testing may not be realized. A skilled test team would have knowledge in the following areas:

- Web environments
- Embedded environments
- General test considerations (knowledge e.g., ISTQB and skills such as those outlined in the AST skills guide
- Hardware understanding
- Systems thinking
- Network communication understanding
- Performance test experience and
- Experience with testing other quality characteristics associated with the IoT context.

Likely no single person will have all of these, so a diverse team of experienced and new testers will be optimal. Additionally, training, on-the-job learning and mentoring should be included as part of the project test plans.

Summary

IoT has been around for years, but lagged in usage behind the Web/PCs and smart phones. Now there are indicators that the sheer number of devices and software in IoT is growing rapidly day-by-day. This means that more testing and testers will be needed to minimize the bugs in the IoT devices being released to consumers. This article has introduced some of the problems IoT testers many face, and made some high-level recommendations for testers in the area of test strategies and planning. Like all test contexts, there is much more to this subject. More work on strategies, planning, error taxonomies and tools for IoT is needed.

About Jon

Jon is a senior software person with a M.S. degree in computer science with specialization in software engineering and testing from Colorado State University and B.S. Degree in Math with specialization in civil engineering and software from Metropolitan State College of Denver, Colorado. He has experience in the software domain of real-time, reactive embedded control systems and mobile smart devices as well as test software development using numerous languages. He has over 100 publications and presentations on software reliability, testing, test tools, formal methods, and critical-systems.

Cover Story

Software Testing's New Frontier - Part 1

What you need to know for testing in the new paradigm

By Michael Hackett

This two part article analyzes the impact of the Internet of Things (IoT) product development on traditional testing.

Part one of this series starts with a wide view on the IoT, embedded systems and device development aspects of testing. Part two, to be published in the September issue, will focus on mobile: connectivity, data, security, performance and remote control—commonly from a smart phone.

Embedded systems have been around a long time, and consumers have had internet connectivity for over to two decades, however the explosive growth of internet enabled devices is just in its infancy. Ubiquitous computing is happening now on a large scale.

The testing challenges that are arising out of this explosive growth are very intriguing. Testing roles are changing. People who were trained as traditional testers, working on well understood systems—test engineers—are being tasked with testing a flood of devices on unknown or new platforms. Due to rapid change, acquiring the skills, knowledge and strategies comes from on-the-job training so you have to take what you know and adapt it to the situation at hand.

By traditional software test teams, I mean teams that are made up of a mix of technical testers and subject matter experts, black and gray-box testers who are typically unfamiliar with testing during hardware development; all of whom will need to adapt rapidly to new platforms, new test types and build new test skills.

The risks involved in testing the IoT can be much greater than traditional application testing. There are apps being developed for devices that connect to other devices and/or systems across the internet, which opens avenues for failures. If you miss or discount a bug, it can cause a ripple effect, and your company may face significant liability.

The systems that make up the IoT are very complex. New and more intelligent sensors are produced every day. Just a few years ago, the hardware sensors and device did all the work. Now, estimates are that software does more than 50% of the work on embedded systems; that is a big shift.

For the reasons mentioned, I will focus on test issues and strategy as it applies for testing the IoT piece rather than on embedded system testing piece. Embedded system testing is well-understood, and there are many sources of information already published on it.

A strong test strategy: *Your test strategy must be effective to be successful.*

Arnold Berger of the University of Washington points out in *The Basics of Embedded Software Testing*, "Many studies (Dataquest, *EE Times*) have shown that more than half of the engineers who identify themselves as embedded software and firmware engineers spend the majority of their time fixing embedded systems that have already been deployed to customers."

This is a startling piece of information to me. Is this because of poorly planned projects; no attention to quality during development, or is the reason simply not knowing how to test these types of systems? Clearly, any IoT or embedded project has to include a great testing foundation or you may be doomed to becoming an expensive support person.

To get started, you need to have a great testing practice in place. Testing processes and practices must be right on target to have any hope of executing an effective testing job. Clear requirements, detailed user stories, unit testing, continuous integration, lean test plans, coverage measurements, great communication, etc.- all need to be part of your regular development process. Programmers must practice designing for testability—writing callable test hooks into the code—in order to benefit the entire product team. Good programming practice and team processes will go far in releasing a higher quality, safer, more secure product.

Your *regular test strategy* is a good place to begin. Validating functionality, installing upgrades, building smoke tests and regression suites will make sure these are the very best they can be will help verify the product can do what it is intended to do.

Testing is easier to do if you have behavior models since a lot of devices have limited or no UI, and many are total black-boxes. Behavior or state models, and even object diagrams will help plan your testing.

Failure and error testing in this new environment requires more focus than typical application test strategy. Forced error testing, where you inject error conditions into your system to check for proper handling, recovery, and where needed; messaging, all need to happen, not only on the software but also on the hardware. Failover, DR (disaster recovery) – already part of a good test strategy— will grow in importance with the addition of testing hardware failures.

Unlike typical applications, your mix of automated and manual testing may not just be dictated by your skill level and tool. There will be situations that can't be adequately tested with manual processes. Variations in models, sub-models, software versions, and configurations will complicate testing and test automation.

New platforms and the need for tools: *embedded system platforms do not have the tool support you may be used to.*

Most often embedded systems—traditionally stand-alone—have had unique, one-off, home-

grown, *kluged* systems and architecture. Then a few industry leaders began to emerge. Having a stable platform leads to a good IDE (integrated development environment) with developer tools, easily available knowledge about the platform and its limits, recommendations, etc. WindRiver created what has become a hugely successful embedded platform. But now the flood gates have opened. Apple wants iOS to be the platform of choice for home and health IoT devices and Google obviously wants it to be Android. Microsoft has had an embedded software platform for years that has tool and information support, and integration into all other Microsoft solutions. Still, many devices have unique and

not-well-known environments. This can lead to marginal validation and testing of the hardware and little effective gray-box testing.

Without common platforms, tools will be scarce-especially *QA* type test tools as opposed to programmer tools. As we know from the recent growth in smart phone platforms, development of test tools lag. Lack of tools and under-the-covers access hurts the test effort.

Since many of the devices of the IoT have limited or no UI, traditional testers cannot rely on taking matters into their own hands to exercise and stress a system. Somehow you have to get consoles, viewers and simulators to get access beyond the black-box. You will need tools, from memory meters to logs, to code tracers and automation, or your test effort will be severely hampered.

It is crucial that you make your tool needs known to the team. The tools you normally use in your regular test process are a good place to start for testing the devices as well.

Platform and Environment Knowledge for Gray-box Testing: *gray box testing is the most effective testing but you need to information about how things work.*

The most daunting aspect of this new frontier for most test teams is trying to understand the architecture, the OS and its nuances, dive into 3rd party hardware, apps, firmware, understand new connectivity protocols and hardware device limitations as fast as possible. This is all necessary in order to design the most effective test cases. Even then you hope the things you don't even know about the system will not bite you.

Gray-box testing is focused between the code and whatever black-box interface your product has, aided by whatever information you can get of the system. Error guessing is a long-standing method in testing, but in many cases, it is difficult to guess where and what errors may be lurking with little-to-no information on how the system works.

The more information you have, the better you will test. So, gather every document you can; read, read, read. Teach yourself new technologies, and share new information among other testers and your whole team.

It will also be necessary to ask a lot of questions: what about the software is unique, special, newly written or re-written? What interaction do the sensors have with each other (M2M)? What protocols does the device use to talk to the remote control? To other devices? To cloud APIs? What concurrency situations can be set up? What race conditions are possible and Impossible? Which

are going to happen every day? Which are never supposed to happen—ever? Your questioning and information seeking ability will be the key to great bug finding.

Real Time, or Real Time Operating System: *RTOS has unique performance standards and functionality and demands for testing on real device rather than simulators.*

Real-time systems are unique in that the functionality, messages or events are ultimately time sensitive. Many are safety or mission critical systems where a few milliseconds can mean the difference between life and death. Safety critical systems, from medical devices to anti-locking brakes in cars, to house alarms; need superfast response time.

Devices for used for financial and commodity trading services—where seconds can mean a profit or loss of billions of dollars—may need to respond in tenths of seconds so that the entire system will respond in seconds.

Real time systems need higher levels of reliability than typical applications and even typical embedded devices. Special test suites need to be designed to test "critical sequences", the scenarios or sequences that cause the greatest delay from trigger to response .

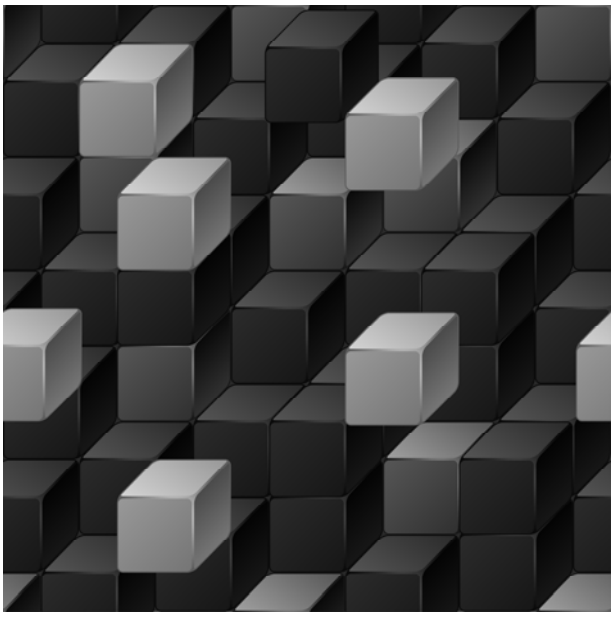
These systems always have unique scheduling routines that need to be verified in addition to race conditions, error handling and concurrency tests. There may also be queues, buffers and varying memory availability that need to be tested.

Acceptance testing clearly has to happen on actual devices. Simulating any part of a critically time sensitive environment will not give realistic results. This does not mean simulators are not useful on real time systems, it just means that simulators are great for testing early but do not replace testing on the actual device.

Systems connected through the internet complicate things. Normally with real time systems there are bandwidth issues, but usually not power issues. However, the internet opens up performance and interoperability problems that need to be overcome.

You can test a home alarm system calling the police. You can also test the home alarm system calling the police with the air conditioner and microwave and clothes dryer on. You can also test the home alarm system calling the police under a power flux as well as with 3 or 4 people in the house streaming movies. This might be a base test and get more complicated from here.

Creating these types of tests requires great test design skills and very clear benchmarks from the team as to the service level and performance tooling skill.



The benchmarks for real time system tests include agreements from sales, marketing, legal departments or regulatory compliance which have to be validated.

Parallel Hardware, OS and Software Development: *concurrent development projects needs great communication and a lot of re-testing.*

A part of the embedded systems lifecycle that traditional desktop and web teams will find very different is the development of the device itself. (article continues on page 20)

Are you *in the Loop*?

When you can't easily or quickly test directly on a system, you need a solution. Sometimes the solution isn't pretty, but it gets the job done.

"Early in my testing career, I tested some mobile devices—what were known as personal information management systems—back in the day before we figured out how to add wireless connectivity and cellular to turn them into smartphones. In the proof-of-concept development phases, before all the parts were fully ready, we needed to test. We had to breakdown and simulate individual parts of the system with accessible and testable prototyped parts. We didn't know it at the time but that was In-the-Loop testing. Each in-the-loop test method addresses a specific group of problems that occur in the development of embedded systems and each brings certain benefits."* Michael Hackett.

Software in the loop - SIL

"Our first testing was narrow scope, there was very limited functionality, and very cheap bugs—and it was done completely simulated on the desktop: software in the loop "

Software in the loop (SIL) testing evaluates functions of code in simulation on the host machine. As in model testing, input test vectors can come from requirements or other models in the executable specification. SIL tests typically reuse the test data and model infrastructure used for model testing in simulation.**

Processor in the loop - PIL

"Next, as prototype hardware was ready, we added processors-in-the-loop (PIL). We got plywood and a few boards and nailed or glued on a few chips and soldered on wires to be able to connect to it. That was our processor prototype."

Conceptually, processor in the loop (PIL) testing is similar to SIL. The key difference is that, during PIL, code executes on the target processor or on an instruction set simulator. The data passed between the model and the deployed object code use real I/O. The SIL model can be reused as a test execution framework for the processor board. With PIL, tests are executed with the embedded algorithm compiled and deployed on the target processor board and with the existing executable specification.**

Hardware in the loop - HIL

"Later as the hardware got closer to being ready, the device hardware grew into mocked up form factors and we tested on versions of software running on desktop simulators running through and tethered to the real hardware devices (HIL). Finding issues early and isolating them sped up development significantly."

Such a configuration reduces the risk of testing on actual, and often expensive, devices. Hardware in the loop is typically done in the lab as a final test before system integration and field testing.

The methods mentioned above can't verify the real time aspects of the design because the simulation and communication overhead with the target board does not allow for real time testing of the algorithm.**

*for the sake of history, these were handheld devices from the pioneer and leader at the time, Palm Computing.

**newelectronics.co.uk, *How in the loop testing aids embedded system validation* http://www.newelectronics.co.uk/electronics-technology/how-in-the-loop-testing-aids-embedded-system-validation/28148

Hardware can be in development while a different team works on the OS and perhaps firmware, with different teams making "software" applications, connectivity, interfaces, API calls, databases, etc.. All of this can take place in parallel, and without a lot a information sharing.

If you are new to this area, it is more common than you would think that test teams from various parts of the product do not know or see each other much. They may not share much information, and likely have very different skill sets. This lack of collaboration has a large impact on testing. Improving communication and sharing knowledge are obvious areas to incorporate into your processes to improve testing. .

Software teams can find they are building code for a moving hardware target. In my experience, the hardware teams are king. They are in control of whatever gets included or left out. When the hardware product is done, the software teams often have to adjust to the new hardware and re-do all the tests. Very often, the software had to be adjusted to make up shortcomings of the hardware.

It is pretty much the same with the system or OS. Whatever the OS team includes is *it*. The *software* or apps teams, usually the last teams in the process, might have to readjust to the new target and re-run all the tests as though for the first time. This does not simply mean re-run a regression suite, it may require re-doing exploratory testing, error guessing and all the tests on the new hardware and OS.

Software teams, can't wait to schedule their work until the hardware and OS team is done—nor should they. Software teams often find bugs, issues or limitations using unfinished functionality on beta stage hardware and OSs that the hardware and OS teams did not catch.

Test Automation: *Diverse test needs, Lack of tools, testing on simulators or through consoles complicates test automation*

Varieties of configurations, versions, patches, updates and supported devices and platforms makes automation mandatory and complex. Finding a tool specific to a platform may not be possible, so customization is essential.

Emulators are very useful for device automation. However, a simulator is not the same as a device. If all the automation is only on a simulator a lot of manual testing will have to be done on the actual device.

As always with automation, test design is the primary key to success. Every type of testing we have covered has unique flavors of automation. Databases, install and upgrade, interoperability, connectivity, performance, security—all have different needs for successful test automation independent to *functionality* validation and testing.

Summary

There is no magic answer for how to test the IoT. It is complicated with many unknowns, but it is also exciting., Adding internet connectivity, to embedded systems will build skills to take you far into testing in the 21st century. Seek information. Build skills in a variety of test types, platforms and tools.

It's currently a kind of "Wild West" mentality in this blossoming industry with few standards. Many platform providers have little real focus on performance, security and interoperability. This will undoubtedly change over time. But for now – you are testing in uncharted waters.

Test early, test often. Report risk and coverage limitations even more than you report what you have actually tested.

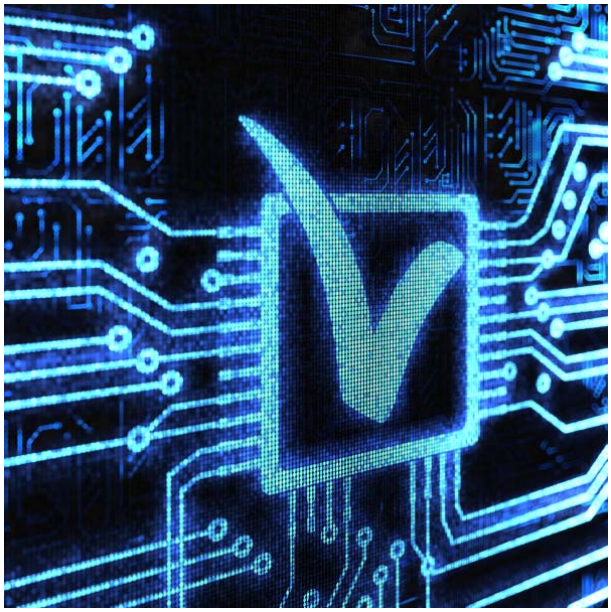
Remember, in part two, we will be investigating more mobile internet considerations such as, remote control, performance testing, security testing, cloud APIs, Big Data testing and interoperability testing.

Blogger of the Month

Why you Need a Software Specific Test Plan

Experience-based recommendations to test the brains that drive the devices

By Philip Koopman



In essentially every embedded system there is some sort of product testing. Typically there is a list of product-level requirements (what the product does), and a set of tests designed to make sure the product works correctly. For many products there is also a set of tests dealing with fault conditions (e.g., making sure that an overloaded power supply will correctly shed load). And many companies think this is enough .. but I've found that such tests usually fall short in many cases.

The problem is that there are features built into the software that are difficult or near-impossible to test in traditional product-level testing. Take the watchdog timer for example. I have heard in more than one case where a product shipped (at least one version of a product) with the watchdog timer accidentally turned off. Just in case you're not familiar with the term, a watchdog timer is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". (Wikipedia)

How could this happen? Easy: a field problem is reported and the developer turns off watchdog to do single-step debugging. The developer finds and fixes the bug, but forgets to turn the watchdog back on. The product test doesn't have a way to intentionally crash the software (to see if the watchdog is working) so the new software version ships with watchdog timer still turned off, and the device doesn't recover without human interaction. That's a problem if you're building, let's say, a Mars rover.

And..well, here we are, needing a Software Test Plan in addition to a Product Test Plan. Maybe the software tests are done by the same testers who do product test, but that's not the point. The point is you are likely to need some strategy for testing things that are there not because the end product user manual lists them as functions, but rather because the software requirements say they are needed to provide reliability, security, or other properties that aren't typically thought of as product functions. ("Recovers from software crashes quickly" is typically not something you boast about in the user manual.) For similar reasons, the normal product testers might not even think to test such things, because they are product experts and not software experts.

So to get this right the software folks and product testers are going to have to work together to create a software-specific test plan with the software requirements that need to be tested, even

if they have little directly to do with normal product functions. You can put it in product test or not, but I'd suggest making it a separate test plan, because some tests probably need to be done by testers who have particular skill and knowledge in software internals beyond ordinary product testers. Some products have a "diagnostic mode" that, for example, sends test messages on a network interface. Putting the software tests here makes a lot of sense.

But for products that don't have such a diagnostic mode, you might have to do some ad hoc testing before you build the final system by, for example, manually putting infinite loops into each task to make sure the watchdog picks them up. (Probably I'd use conditional compilation to do that -- but have a final product test make sure the conditional compilation flags are off for the final product!)

Here are some examples of areas you might want to put in your software test plan:

- Watchdog timer is turned on and stays turned on; product reboots as desired when it trips.
- Watchdog timer detects timing faults with each and every task, with appropriate recovery (need a way to kill or delay individual tasks to test this).
- Tasks and interrupts are meeting deadlines (watchdog might not be sensitive enough to detect minor deadline misses, but deadline misses usually are a symptom of a deeper problem).
- CPU load is as expected (even if it is not 100%, if you predicted an incorrect number it means you have a problem with your scheduling estimates).
- Maximum stack depth is as expected.
- Correct versions of all code have been included in the build.
- Code included in the build compiles "clean" (no warnings).
- Run-time error logs are clean at the end of normal product testing.
- Fault injection has been done for systems that are safety critical to test whether single points of failure turn up (of course it can't be exhaustive, but if you find a problem you know something is wrong).
- Exception handlers have all been exercised to make sure they work properly. (For example, if your code hits the "this can never happen" default in a switch statement, does the system do something reasonable, even if that means a system reset?).

Note that some of these are, strictly speaking, not really "tests." For example, making sure the code compiles free of static analysis warnings isn't done by running the code. But, it is properly part of a software test plan if you think of the plan as ensuring that the software you're shipping out meets quality and functionality expectations beyond those that are explicit product functions.

And while we're at it, if any of the above areas aren't in your software requirements, they should be. Typically you're going to miss tests if there is nothing in the requirements saying that your product should have these capabilities.

About Philip

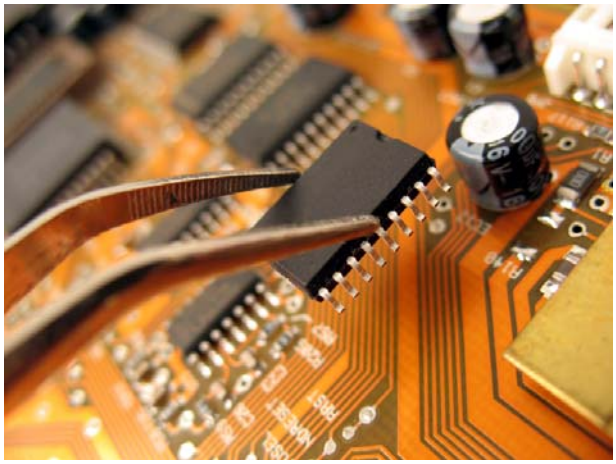
Philip's background includes time as a submarine officer for the US Navy, a principal in a couple small startups, an embedded CPU architect for Harris Semiconductor, and an embedded system architect for United Technologies Research Center. At Carnegie Mellon Philip has worked in the broad areas of wearable computers, software robustness, embedded networking, dependable embedded computer systems, and autonomous vehicle safety.

Feature

Proven Practices for Testing Embedded Software

Build the right test platform including infrastructure, virtual lab and process

By Andrey Pronin, Auriga



Testing embedded software is both similar and dissimilar to application software testing. The first eye-catching thing is that embedded software is significantly less visible to the end user. User interfaces are limited; there may be a console-based text menu, a simple command line interface, a set of digital inputs of outputs, or something similar, but rarely do we get more than that. On the other hand the inter-component interfaces can be very rich and complex—including APIs to the higher-level software, implementations of various communication, data exchange, control, and other standards, etc. Thus the main focus of embedded software testing is not on testing the user interfaces, but on testing the components not visible to the end users.

The second major difference is the level of the dependence on the hardware specifics. Embedded software is the level of the software closest to the hardware. Other software types such as operating systems and applications may be built upon the interfaces provided by the embedded software such as BIOS or boot loader. The embedded software itself, even if it uses some more or less standard framework underneath, needs to care more about hardware details. Embedded software by definition is designed for a particular hardware unit (or a set of hardware units in common case). Often, those hardware units are developed in parallel with the embedded software. The created software is the first to run on it.

Unlike application development, in the embedded world we can't rely on the fact that the operating system is already tested on the hardware platform, or that the ability of the hardware itself to execute various software has already been thoroughly tested. As a result, the developed software may have solutions and workarounds specific for particular hardware revisions.

Operation of the embedded software may depend on such things that we usually don't care about for application-level software, like the length of the cable, type of the mouse, serial port frequency, or type of the devices connected to the same bus that makes the successful execution of embedded software degrees more dependent on the particular hardware unit and on the behavior of the other units in the same bus or network.

Compared to conventional cases, race conditions are mostly caused not by the interaction of the internal software components, but rather by the interactions of the software with the environment. So, the number of factors and parameters that can influence the operation is bigger than for the average application. And reproduction of a defect is more difficult. Support operations, such as software deployment, upgrade, getting debug information, also differ from what we usually see in conventional application-level software with plug-n-play concept, installation wizards, ability to attach a convenient debugger from one of the IDEs, or at least dump all debug output lines to a large file on disk.

In the embedded world we often need to put the software in a special mode, disable EEPROM write -protection, attach to some file-distribution (like TFTP) server, reboot a couple of times, and care about other similar things. That makes the software update process lengthy and inconvenient. And, it might be that the device that stores your software supports only a limited number of re-write cycles.

During active development phase, software versions tend to be updated less frequently than for the other forms of software. New revisions are typically deployed only after a significant number of defects are resolved. Thus, the testing process should attempt to find as many defects as possible, and not stop after the first one, even if it makes the product crash.

Embedded Software Testing Challenges

The specifics of the embedded software domain imply certain requirements for the organization of the testing process. The focus on non-human interfaces leads to the fact that, we can't use a manual interface testing approach.

To test the developed embedded software, we first need to develop special applications—test agents—need that provides stimulus and captures response through the non-human interfaces. It is also often necessary to emulate particular electrical signal patterns on various data lines to test the behavior of the embedded software for such inputs. It can be done using special hardware/ software complex along with a built-in special test agent to control that complex.

A high level of hardware dependency and the fact that the embedded software is often developed in parallel with the hardware leads to several important consequences. First, there may be only few samples of the newly developed hardware. Second, the range of the hardware unit types to test the software on can be quite wide. Thus, typically the testing team has to share a very limited set of hardware units among its members and/or organize remote access to the hardware. In the second case, this means that the testing team has no physical access to the hardware at all.

Another aspect of having the software developed for a freshly created hardware is a high ratio of hardware defects that can be discovered during the testing process. Any discovered defect may be related to either the hardware or the software. Always keeping that in mind is especially important for embedded software projects. What's worse, the software may work just fine with one version of the hardware, but not so well with another.

We have already mentioned that defects are harder to reproduce in the embedded case. That forces the embedded testing process to value each defect occurrence much higher than in a conventional case and attempt to gather as much information as possible rather than simplify looking for the root of the defect. That, combined with the very limited debug capabilities of the embedded products, gives us another challenge.

Limitations related to software updates requires persistence with the testing process to discovering as many bugs as possible for a given software revision. It also increases the importance of build and deployment process.

A high level of requirements on the robustness/ availability front leads to the need for very thorough stress testing. A consequence of that fact is the need to emulate the sequences of rapid-follow events to check for race conditions under those circumstances.

Automated vs. Manual Testing

First of all, it is obvious that using manual testing as the main method for embedded testing projects is very difficult, if not impossible. Routine, time-consuming, repetitive stress testing, working

with non-human interfaces, the need to discover race conditions for fast-sequence events, and a host of other factors all complicate the task. Thus automated testing is a cornerstone approach.

Of course there will always be a percentage of tests that is more cost-effective to run manually than automate. But that percentage is smaller than usual, dictated by higher relative efficiency of automation in remote access environment (the alternative to which is organizing a trip to the remote lab) and special supporting means described later. In any case, automation is typically done for more than 95% of the test cases. Having stated that, it important to understand that automation and usage of test agents doesn't simply change the way of executing the test cases and presenting results, it affects all aspects of the testing process.

Test Design and Tracing Requirements

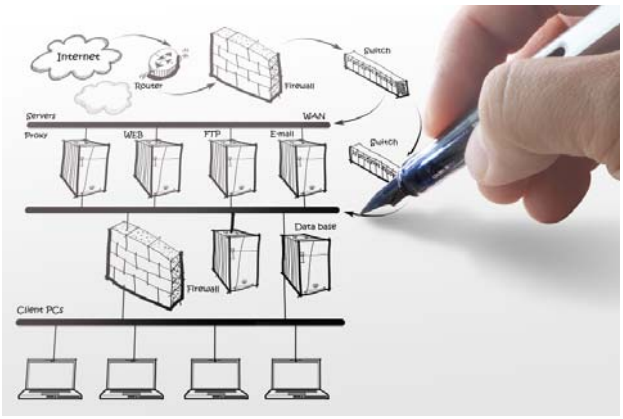
Two things must be understood. First, a great number of the test cases created for the embedded software simply cannot be executed manually. Thus a straight forward test design approach—get requirements; design test cases; run manually; optimize; fix; detail; create script based on the manual case—doesn't work here. Second, unlike regular methodology, the software requirements specification does not lead to, and is not traced to, just the set of the test cases.

Instead, based on the software requirements of the embedded software, two artifacts are created—the set of the test cases and the requirements for the test support infrastructure which consists of the automation framework and test agents. In the formal sense, the embedded software requirements are traced to the test cases, which in turn are traced to the software requirements for the agents and framework. But from the practical perspective, test cases and support software requirements cannot be separated.

Validation of The Test Support Infrastructure

The second influence on the testing process is in the fact that the support software must itself be validated. Basically, that means that first, the test agents and the automation framework must be tested themselves—test design, execution, coverage analysis, and all other activities are performed for them as well. Test agents are typically relatively simple software entities with a limited set of requirements, so testing them is significantly simpler than testing the original software product. Still, they often need to implement complex data exchange protocols (including encryption, authentication, compression, connection establishment, and what not), so testing them is not at all simple.

Complete testing of the test agent is often impossible without having more-or-less, a working version of the target process. So, passing tests for a test agent also means passing basic functionality tests in a particular area for the target software.



During this testing, previously verified test agents and hardware debugging tools—bus analyzers, network sniffers, JTAG probes, and oscilloscopes—are extensively used. The hardware debugging tools are especially useful at this stage of achieving a basically functional application. This has another natural implication on the embedded software development process. The design of the test support tools is done parallel with the target embedded software design, and the development plans for the target software and test agents are highly dependent.

The second component of the test support infrastructure, automation framework, also obviously requires validation. However, unlike the test agents, which perform functions specific to a particular embedded product, it can, and should be, designed and implemented as project independent, at least inside some wide technological or organizational segment. That saves a great amount of testing effort and doesn't need to be repeated for every next project.

Defect Tracking and Analysis

Besides the direct verification and validation effort, the need to validate the test support infrastructure also influences the defect lifecycle and defect tracking repository setup. For embedded software several possible origins should be considered for each defect: the target software, the underlying hardware and the test support infrastructure. One example of the practical consequences of that leads to specifying target software, hardware, and test support suite IDs in every discovered defect record. Another example is including the representative of the test support infrastructure development team in the triage committee for the project.

For hardware-caused defects, the testing team must include a person with hardware engineering skills along with knowledge of using various hardware debugging tools mentioned above. This person should also be included in the triage committee to examine each defect from the point of view of probability for it to be of hardware origin. This person will also provide guidance to the team regarding the suspicious signs in hardware behavior and gather additional data for analysis if a hardware defect is suspected.

Hardware Coverage Matrix

A higher probability of a hardware defect doesn't lead just to the need to specify hardware ID in the defect record and having a hardware engineer on the team. The target software must also be tested on the range of the possible target hardware types and revisions. That doesn't mean that each test case must be run on all possible hardware units/types/revisions. A conscious choice between the coverage and cost/time must be made. It is often possible to combine hardware units in groups for testing each functionality area, or at least perform random selection for regression testing purposes. The test strategies defined for different projects may vary in this aspect based on the project constraints and requirements.

In any case, a hardware coverage matrix is required. All "test case and hardware unit" combinations that should be verified are marked in this matrix. For obvious reasons, automation framework should allow for specifying and changing the matrix without affecting the bodies of the individual test cases.

Software Builds

Establishing the right build and deployment process is also essential for the success of the embedded software testing task. It is important to correctly identify the target software revision, for which a defect is revealed. Several techniques are used to address the issues related to the software build identification.

One of the useful practices is obtaining the build number from the running target software at the beginning of the test suite execution—the embedded software that has some user interface often allows getting that information. Using this practice prevents incorrect identification of the version in defect records, if a test suite was run against the wrong version by mistake.

Another practice is used for the smoke tests of regular software releases. According to the practice, the test support infrastructure contains all necessary tools for making the build, assigning it a unique number, tagging the source tree, archiving the binaries, transferring the binaries to the deployment server (e.g. TFTP server) and then to the target board, and updating the software on the board. Such operations may be performed at the beginning of the overnight smoke test for a regular build. For the projects with no limitations on the number of software updates for the target hardware unit, this operation can be performed completely (build and deploy on the board) or partly (deploy only) before every build to ensure the right version to be used during the testing.

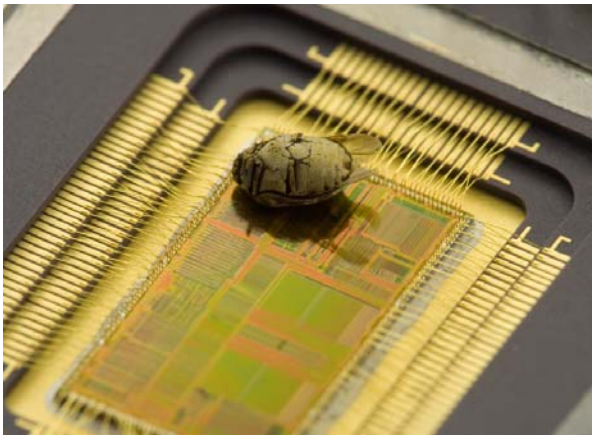
Debug Support

One of the goals of the good testing process, besides revealing as many defects as possible, should be assistance to the developers in resolving the defects. A defect found by the testing team that can't be reproduced by the development team and thus can't be fixed due to insufficient information provides little value.

As stated, in the embedded world the defects are harder to reproduce, thus as much information is possible should be gathered on the first occurrence. Due to the fact that debugging is also more difficult for the embedded software, the develop-

ment team often uses special debug builds or special debug modes of the target software with increased logging capabilities.

There are two implications of this situation for the testing process. First, the timing and other characteristics of the debug and release versions of the target software may differ, and the defect seen on one version may never be seen in a different version. Thus it is important to keep track of the software revision, for which the defect was discovered by testing.



Second, the test cases should be designed to allow using these extended capabilities of the debug version or mode. When a defect is revealed the test case should store the debug output of the software in the test log tied to the test result, so that a developer assigned to resolving the defect can use this data during the analysis. The test case should also be able to detect the type of version of the target software—debug or release, or switch between the modes. The details of that are highly project-specific and are usually implemented either through the parameters passed to the test case, or by employing a specialized test agent.

Test Runs

Due to the contradicting characteristics of the embedded software product, there are two types of test runs employed for it.

An ideal method is to batch-run test cases. All selected test cases are run according to the hardware coverage matrix, and the results are stored in the test log. If an error is detected, the test run doesn't stop, but rather all possible information about the system state at the time the defect was discovered (and all debug support techniques are important here) is captured, and testing continues with the next test case. Needless to say, the test support framework should perform a complete clean-up after each test case to avoid influence between the test cases in general, and a series of failed cases after the first crash in particular. Such clean-ups often include system reboot, typically software reboot after a successfully completed test case, and a hardware reboot after a failure.

Such test runs are lengthy; the required time further increases the need to clean up. Due to the length of time these runs are typically scheduled to be performed in automatic mode overnight. Such batch runs are especially useful as smoke/regression tests for new builds.

In certain cases tests are run until the first failure. Then the test run is stopped and the system state is preserved. A developer is then notified and allowed to examine the system state in detail to reveal the root cause of the failure. It is also possible to create an automation framework that would break the test run only if the failure occurred in a particular test case (or a set of test cases). Such test runs are useful for hunting down defects, for which information gathered in the batch mode is insufficient and a developer needs to get access to the system at the moment of defect to investigate it.

Virtual Laboratory

The methodological approaches described in the previous sections allow forming the testing process relevant to the specifics of the embedded software testing. However, there is another important part of the approach—a software and hardware solution, called Virtual Laboratory, or VL. This solution provides the means for solving several technically complex problems faced during testing.

First, it contains a database of the existing hardware units. The units are identified by simple string IDs. For each unit, it is possible to specify several properties, such as hardware revision, communication IDs—IP address, MAC address, login credentials, etc. For a test script this means that by passing a unique unit ID as a parameter, it can restore all other parameters that are required to communicate with this board and provide complete defect reports.

Second, a VL supports serial consoles, power bars (devices allowing switching the power on and off for the target units), and dry contact controllers (relays). Console/relay/ power bar lines are associated with a particular unit in the hardware unit's database and as a result, all operations with a particular unit are performed by the test scripts based on the name of that unit.

Third, a VL provides a means for ensuring exclusive access to the shared hardware. Before accessing a unit's console the test script must first 'lock' that unit using a special command. While the unit is locked, no other entity can 'lock' it. After all testing actions are performed, the test scripts 'unlocks' the unit, allowing others to control it. Such exclusive locking mechanism prevents interference of different test scripts and human operators attempting to run test on the same board simultaneously.

A VL provides human-friendly command-line interface over secured connection, and can be used both by test scripts and human test operators. A VL serves the base for executing all automated and manual tests for the target software.

Summary

Testing of embedded systems can be quite complex and defects can be quite hard to produce. Taking time to create the right test environment and then creating the right level of testing will help produce the very best test environment.

About Andrey

Andrey has a Ph.D. in applied mathematics and has spent over 20 years in various roles related to software development. He is currently SVP of Strategy, Technology and Marketing at Auriga, and a board member of RUSSOFT, association of software development companies. Auriga is a leading software engineering services provider (www.auriga.com)

The Top 10

Automation Issues to Solve for Embedded Software Testing

1. **Physical access to the embedded system to play tests or get results.** Sensor and other hardware interfaces may need special case access.
2. **Support for test automation in the product itself.** Hooks or callable APIs may need to be added to the code.
3. **Behavioral verification.** Recompiling embedded code on a PC often affects the runtime behavior.
4. **Hardware availability.** At least run verification tests directly on hardware. Automation on a simulator does not replace testing directly on the device.
5. **Security**. Any testing (or agent) code in the embedded system, in particular in a production version, should not open a gate for hacking.
6. **Check for timing issues**. Some devices have time dependent controller software.
7. **Availability of expert team members**. Teams need people who both understand the technologies and are versed in testing.
8. **Automating multi-media aspects**. Sounds being played or LED flashing needs to be tested.
9. **Memory constraints**. There are situations where available RAM may be too low causing system failure.
10. **Non-determinism of the system under test**. Due to any number of reasons (race conditions, random number generators, state) a system may behave differently in different runs making pass/fail criteria and test coverage difficult.

LogiGear

Testing Embedded Systems Isn't Always Fun n' Games

We can help

Test Automation
Consulting
Training
Expert Assistance

See a few of the clients that have relied on us for embedded system testing
http://www.logigear.com/about-us/our-work/by-technology.html#embeddeddevices

Feature

Team Based Testing

Getting everyone involved lets you shorten the loop by testing early

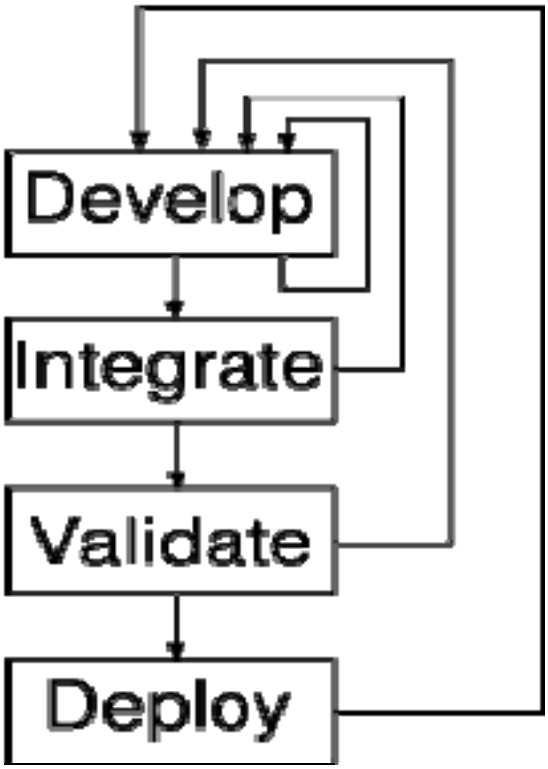
By Arnout Vandecappelle, Essensium/Mind

Developing software for an embedded system often carries more risk than for general purpose computers, so testing is extremely critical. However, there still has to be a good balance between time spent on testing and time spent on development to keep the project on track. As consultants for embedded open source technologies, we at Mind encounter many different approaches to testing with our customers. This article structures these varied experiences and combines best practices and techniques , with a focus on embedded open source software.

The Efficient Software Developer Uses Testing

We develop software because we want to make a working product. Therefore, validation is an essential part of the software development process. Validation should be done from a user's perspective. That makes the loop back to development very expensive: it has been a long time since the code was written so the developer may have to refresh their own memory or the original developer has moved on. Either of these may make it difficult to pinpoint the cause of a problem because everything is already glued together, and there isn't much time because the release is due soon. To tighten that loop, the software should be tested as soon as possible, during development and integration.

Loops back to development exist not only because of validation, but also because the software evolves over time: features are added, requirements shift, supporting libraries are upgraded, etc. All of this results in modifications to the existing code. Unfortunately, every modification may mean that something that used to work, now breaks. This is why agile methods stress testing so much: in agile methods, modifying existing code is much more important than writing brand new code. Pre-existing, automated tests reduce the threshold to modify code. They have to be automated to some extent, otherwise the threshold to actually run the tests becomes too high.



An agile team based approach to testing improves efficiency. By working as a team, developers and testers can shorten the loop by performing early testing. Here are some guidelines to follow.

- Make sure there is a test infrastructure from the very start of the project. It doesn't have to be much, but if nothing is there it becomes increasingly difficult to

create the infrastructure while the project grows.

- Make sure that every team member can run the tests. The easiest way to achieve this is to automate the tests.

- Make sure the tests run fast. That of course means that it can not be very complete. Complete testing is the responsibility of integration and of validation. The software developers, on the other hand, should run the tests after each change, and certainly before publishing changes to the rest of the team. If it takes a long time to run the tests, they will be delayed, which makes the development loop larger. Also it would delay publishing of changes, which makes the integration loop larger.

- Tailor the tests to your implementation. While developing, you know pretty well where the risks are of doing something wrong. For example, when doing string manipulation in C, the main risk is doing something wrong with the terminating 0 byte. Make a test that checks this specifically.

- Distinguish between specific tests and smoke tests. We only need to test the things we are currently modifying. Modifications can break things in two ways: it can break the existing features of the functionality we're modifying, or it can break something unrelated (or expose an existing bug). For the first, we just need to test the functionalities that we're modifying. This typically corresponds to a unit test, but it can be more on the integration level (when modifying the interface between modules, which happens quite often). For breaking unrelated things, those are very often catastrophic (e.g. stack overflow, double free). Therefore, it is often sufficient to check that the system as a whole still works. For embedded systems, it's usually sufficient to boot a system with all features enabled and check that it still does something.

Embedded Testing: Test Hardware, Simulation, Timing and Updates

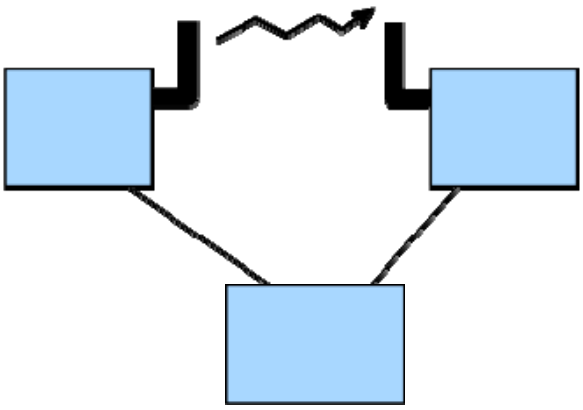
Testing for embedded systems is different than for general-purpose computers. First of all, there is an important hardware dependency, for instance analog audio input, a radio tuner, or a camera. However, the hardware may not be available for an extended time (e.g. there are only 5 boards for 9 software developers). It is often very resource-constrained and doesn't have the CPU power, memory or flash space to accommodate test infrastructure. And its I/O capabilities are usually rather limited, e.g. lack of writable file system for input data or saving traces. These physical limitations can be overcome by stubbing and simulation. Second, it interacts non-trivially with its environment. For instance, a video screen should show the video in real time and degrade gracefully when too many streams are shown simultaneously. These things make up the essential difference between the embedded system and a desktop media player, and are the reason you can't simply use existing software as is. So these things should also be tested. Finally, updating the software once the embedded system has been sent into the field is completely different from updates of general-purpose computers. Therefore special attention has to be paid to the update procedure and it should be tested to assure it is repeatable by the end user.

Testing the Hardware Setup

Since the embedded system software depends on the hardware, it is important to have a good setup of test hardware. This is typically a concern for the validation team however, efficiency can be boosted if the validation team makes test hardware available to the developers as well. A good test hardware setup allows remote control of the I/Os and remote updates of the firmware, so that it can for instance be placed in an oven for testing. An *nfsroot* is a good solution to allow remote updates. Not only control the I/O remotely, but also perform power cycling. This makes it possible to test the behavior when faced with sudden power loss.

As an example, consider testing a wireless metering device. The test setup could consist of two of these devices: one with the actual firmware under

test, the other is a controller that provides radio input and monitors radio output. Both of them are network-connected to be accessible for testing. Another example is an audio processing board, where the (analog) audio inputs and outputs are connected to a PC that generates sine waves and samples the output.



Simulation

To be able to perform testing close to the developer, we can perform simulation. The most obvious form of simulation is using a virtual machine, for instance KVM/qemu or VirtualBox. This allows you to simulate the entire system, including the kernel. This has several disadvantages. First, you will probably need to add new peripheral simulators for your particular device. Creating such a peripheral simulator correctly can be very tricky. Second, the simulators are not entirely reliable (especially when it comes to peripherals). Thus, you may end up debugging problems which didn't actually occur on the system, but emerged in the simulator. Finally, simulation carries a speed penalty. For virtual machines (KVM, VirtualBox), the speed penalty is limited to the times when virtualization kicks in, e.g. when serving interrupts or accessing peripherals. For emulation (qemu), the penalty kicks in for every instruction. However, since the development server often runs an order of magnitude faster than the target platform, emulation may still turn out to be faster than running it on the actual system.

An alternative approach is to run your application code natively on the development host. In this case, you don't try to simulate the entire system, but only the (user-space) application code. To make this possible, you need to add a Hardware Abstraction Layer (HAL) to your application, which has a different implementation on the development host and on the target platform. If you heavily use standard libraries, these often already form a HAL. For instance, Qt and GLib have different implementations depending on the platform they are compiled for. The HAL is in addition a good way to make sure the application is easy to port to new hardware. If the application consists of several interacting processes, it is usually advisable to test each one in isolation. Using e.g. D-Bus for the IPC simplifies this, since you can replace the bus with a program that gives predefined reactions.

Running the application on the development host has several advantages. First of all, you have a much larger set of debugging tools available, including debugger, IDE, valgrind, trace tools, and unlimited tracing. Second, it is often much faster than either simulation or running it on the target platform.

Whatever the simulation approach, it also has to be made reproducible. That typically means that inputs are taken from a file instead of the normal channels (network, A/D, sensors, FPGA, ...). Also outputs go to a file instead of to the normal channels, to allow off-line analysis. Creating reproducible inputs is even useful on the target platform itself, where you can debug the full system including timing.

Timing

Embedded systems show a lot of time-dependent behavior. Part of this is hidden in the HAL (e.g. timeouts of devices), but often also the application itself has time as one of its inputs. For example, a video display unit has to synchronize several streams for simultaneous display, or a DSP algorithm has to degrade gracefully when the processor is overloaded. Also race conditions in a multi-thread program depend on the timing. This time-dependent behavior is hard reproduce, especially when using simulation.

On the target platform, the time-dependent behavior can be approximated fairly well. The only requirement is that the simulation of inputs (see above) also includes information about the time at

which this input is available. The thread that parses the input adds delays to match the timestamps in the input file. If the input timestamp has already passed, this is equivalent to a buffer overflow in e.g. DMA and is probably an error. Clearly, the HAL should be carefully thought out to make this scheme possible, e.g. sizing buffers so they match the size of DMA buffers.

One possibility for making timing reproducible in simulation is to simulate time as well. The simulator keeps track of the simulated time of each thread. Every thread (including the input thread) adds delays to the simulated time; the delays should correspond (more or less) to the amount of processing time it would take on the target platform. Whenever a thread communicates with another thread or with the HAL, a synchronization point is added: the thread blocks until the simulated time of all other threads has reached its own simulated time. This concept was invented by Johan Cockx at Imec.

Updates

Unlike PCs, embedded systems are very easy to "brick", meaning that if something goes wrong while updating the firmware, it is very difficult to recover from that because it's not possible to boot from a USB or CD-ROM. Often, the device isn't even easily reachable, for example, the controller of a radar buoy in the middle of the ocean just has a network connection; if something goes wrong with an upgrade, somebody has to travel in a boat for two days to recover it—assuming they can find it in the first place.

Therefore, for embedded systems it is essential that the update system works and never fails. It is mainly the responsibility of the validation team to test if it works, but the developer has a much better insight in where it can go wrong. This is where a team testing approach has significant benefits and can jointly take into account the following in the update mechanism:

- Power failure in the middle of the update, which corrupts the root file system or kernel. To protect against this, the updated software should be installed in parallel with the existing software. Links should be updated only after successful installation, and this should be done atomically (i.e. using rename(2), not editing a file in-place). Package managers usually take care of this pretty well. Of course, a journalled file system is needed as well to avoid corruption of the file system itself.

- Integrity of the software, which may be jeopardized by e.g. data loss over a serial connection or premature termination of a network connection. Package managers protect against this with a hash and signature.

- Installation of incompatible pieces of firmware. Again, package managers help to protect against this.
- Installation of firmware that is not compatible with the hardware. This is most pressing for the kernel and boot loader, but also other pieces of software may have a strong dependency on the hardware. A package manager can help by creating a *platform name* virtual package and depending on it.

Clearly, package managers help a lot to secure the update system. However, they can't be used on read-only file systems (e.g. squashfs). Other solutions need to be found in that case.

Conclusion

It cannot be stressed enough that testing should start early. Developers can do a lot of testing on their own, but in agile environments, team collaboration can make testing just one of the project tasks. Embedded software has specific constraints, like hardware availability, which make it even more important to think about testing early on as a team.

About Arnout
Arnout is a Sr. Embedded Software Architect at Essensium/Mind. He has extensive experience in embedded system design, with a particular interest in software debugging.

Embedded Systems Glossary

Closed System - In the context of embedded systems this relates closely to the engineering context where every input and every response (or output) can be known and can include a specific time. In addition the software is purposely designed for restricted access.

Open System - Specific systems and or applications that allow unrestricted access by people and/or other computers.

Things that think (MIT) - Computing like devices with programming logic that can determine their own interactions and outputs. These devices can interact with other devices, the internet and the physical environment. Source: *MIT*

Internet of things - interconnection of uniquely identifiable embedded computing like devices with the existing internet infrastructure.

Ubiquitous computing - a concept in software engineering and computer science where computing is made to appear everywhere and anywhere. Ubiquitous computing can occur using any device, in any location, and in any format. The underlying technologies to support ubiquitous computing include the internet, advanced middleware, operating system, mobile code, sensors, microprocessors, new I/O and user interfaces, networks, mobile protocols, location and positioning and new materials. *Source: Wikipedia*

Information appliance - A device that is designed to easily perform a specific electronic function such as playing music, photography, or editing text. *Source: Wikipedia*

Electronic control unit - a central, sometimes distributed but clearly distinguishable, part of a mechanism that controls its operation, for example a computer that controls the ABS of a motor vehicle.

Microprocessor/microcontroller - used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems.

Embedded operating system - an operating system for embedded computer systems. The application, including the operating system, is usually statically linked together and does not load and execute applications.

Embedded software - computer software, written to control machines or devices that are not typically thought of as computers. It is typically specialized for the particular hardware that it runs on and has time and memory constraints. A characteristic feature is that no or not all functions of embedded software are initiated/controlled via a human interface, but through machine-interfaces instead. *Source: Wikipedia*

Firmware - In electronic systems and computing, firmware is the combination of persistent memory and program code and data stored in it. Typical examples of devices containing firmware are embedded systems (such as traffic lights, consumer appliances, and digital watches), computers, computer peripherals, mobile phones, and digital cameras. *Source: Wikipedia*

System on a chip - A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions—all on a single chip substrate. A typical application is in the area of embedded systems. *Source: Wikipedia*

LogiGear®

LOGIGEAR MAGAZINE
AUGUST 2014 | VOL VIII | ISSUE 3

United States
4100 E 3rd Ave., Suite 150
Foster City, CA 94403
Tel +1 650 572 1400
Fax +1 650 572 2822

Viet Nam, Da Nang
7th floor, Dana Book Building
76-78 Bach Dang
Hai Chau District
Tel: +84 511 3655 333
Fax: +84 511 3655 336

Viet Nam, Ho Chi Minh City
1A Phan Xich Long, Ward 2
Phu Nhuan District
Tel +84 8 3995 4072
Fax +84 8 3995 4076